Reflective Metaprogramming in C++

Daveed Vandevoorde
Edison Design Group

Topics/Overview

- Generalities
  - What is reflective metaprogramming?
- C++ Template Metaprogramming
  - Principles
  - Pros and cons
- The Metacode Extension
  - Principles
  - Constructs
  - Implementation notes
Part I

Generalities

What is “Metaprogramming”?

- Meta? *The New Shorter OED:*  
  - “Denoting a nature of a higher order”  
  - “Denoting change, alteration, or effect generally”  
  - ...  
- Programming = creating/modifying a program  
- Metaprogramming =

  Creating a program that creates or modifies another program
What is “Reflection”?  

A program’s ability to observe itself  
• At a sufficiently high level  
  • Inspecting bytes is not the spirit  
• At run time or at translation time  
• Partial (e.g., just types) or complete  
  (including executable code)

Applications

• “Middleware”  
  • Distribution  
  • Persistence  
  • …  
• ABI bridging  
• API transitions/usability  
• Component-specific optimization  
• All kinds of instrumentation
Part II

C++ Template Metaprogramming

C++ Template Metaprogramming Basics

- Use the template instantiation process as a computational engine
- Use parameterized types and constants to record state
- Use explicit or partial specialization to implement conditions
  ♦ Computationally Complete
C++ Template Metaprogramming Example A

template<int B, int N> struct Pow {
   enum {
      value = B*Pow<B, N-1>::value;
   };
};

template<int B> struct Pow<B, 0> {
   enum {
      value = 1;
   };
};

int bitset[Pow<2, 13>::value];

C++ Template Metaprogramming Example B

typedef char One; typedef char (&Two)[2];

template<typename T> One f(typename T::X*);
template<typename T> Two f(...); // Ellipsis parameter

template<typename T> struct HasMemberTypeX {
   enum {
      yes = (sizeof(f<T>(0)) == sizeof(One))
   };
};

struct S1 { typedef int X; };
struct S2 { }

int a1[HasMemberTypeX<S1>::yes]; // OK
int a2[HasMemberTypeX<S2>::yes]; // Error
Plug Plug Plug

C++ Template Metaprogramming Strengths

- Serendipitous
- Widely used
  - Perhaps the only truly successful form of compile-time reflective metaprogramming
C++ Template Metaprogramming Weaknesses

- Verbose
- Indirect/Opaque
- Expensive
- Poor tools/diagnostics
- Limited scope
- Limited reflection

C++ Template Metaprogramming Cost

- Speed
  - Requires complete C++ semantic checking
- Storage
  - HasMemberTypeX<T>: 3.5 KB/instance
  - Pow<B, N>: N x 2.2 KB/instance

(EDG 3.0, Strict ANSI, minimal configuration)
Part III

The Metacode Extension

C++ Native Metalanguage Challenges

• Already a very complex language
  • Arcane properties and restrictions
  • Exposing compiler internals not practical
• Metalanguage must be portable and neutral

```cpp
#include <iostream>
typedef int Int;
void f(Int);
int main() {
    std::cout << typeid(f).name() << std::endl;
}```
The Metacode Extension: Principal Components

- Metacode functions
  - Compile-time function evaluation
- Code injection mechanisms
  - Code generation by metacode
- Standard Metacode Library
  - Lives in namespace stdmeta
  - E.g., is_lvalue(expr)
- Metacode blocks
  - Metacode in declarative contexts

Metacode Functions: General Principles

- Introduced by new keyword metacode
  - But after template parameters
  - Function can be ctor, operator, ... but not virtual
- Calls can be constant-expressions
  - Compile-time evaluation!
- Cannot call non-meta functions
- Arguments of meta-calls:
  - Implicit conversion not performed
  - “Value” of parameters only when constant argument
- Maybe: “metacode ...” parameter
  - Like regular ellipsis but always “best match”
Metacode Functions: Example

```cpp
template<typename T> metacode
T power(T b, unsigned n) {
    T r = 1;
    for (int k = 0; k<n; ++k) r *= b;
    return r;
}

float a1[power(2, 3)]; // OK: Same as a1[8]

int p = 3;
float a2[power(2, p)];
    // Error: Metacode routine attempts to
    // access value of nonconstant p
```

Standard Metacode Library
Metacode Types

- Many C++98 types OK
- Distinct address spaces (pointers, references)
- Implementation currently limited
- `stdmeta::string_literal, stdmeta::id`
  - To manipulate string literals and identifiers
- `stdmeta::type`
  - To manipulate C++ types
- `stdmeta::array<T>, stdmeta::table<KT, VT>`
  - Possibly the only dynamic meta-structures
  - E.g., type lists: `array<type>`
Standard Metacode Library
Built-in Metacode Functions

- Building blocks for user-defined metacode functions
- Often “magical”
  - is_accessable("C::x")
  - in_normal_function()
- May have compile-time side-effects
  - error("Too weird!")

Metacode injection mechanisms

- `metacode->{ <code> }`
  - Injects <code> in enclosing class/namespace scope
- `metacode->::{ <code> }`
  - Injects <code> in global namespace
- `metacode-> N::M { <code> }`
  - Injects <code> in namespace N::M
- `return-> <expr> ;`
  - Injects non-constant expression
- Lookup rules:
  - Same as C++98
  - Variables from metacode accessible as simple, nondependent identifiers translate to appropriate tokens
Metacode Injection: Example 1

```cpp
metacode
double mypow(double b, int n) {
    using ::stdmeta::is_constant;
    if (is_constant(b) &&
        is_constant(n) &&
        n >= 0) {
        return power<>(b, (unsigned)n);
    } else {
        return-> ::std::pow(b, n);
    }
}
```

Metacode Injection: Example 2

```cpp
metacode define_fields(array<type> types) {
    for (int k; k<types.length(); ++k) {
        type FieldT = types[k];
        id FieldName = id("field" +
            string_literal(k));
        metacode-> {
            FieldT FieldName; // Metacode identifiers
            } // translated according
        } // to their types.
    }
```
**Metacode blocks:**
Metaprogramming in Declarative Contexts

- Allows for metacode to appear where no expressions are allowed
- In definitions of classes and functions

```cpp
template<typename T> struct S {
    metacode {
        // Start metacode block
        if (stdmeta::typevar<T>().is_reference()) {
            stdmeta::error("No reference, please.");
        }
    }
    // ...
};
```

**Problems This Solves/Helps**

- Constrained genericity
- Move semantics
- Forwarding problem
- User-defined literals
- Efficient compile-time varargs
- ...

Implementation Notes (1)

- Partially implemented in an internal copy of the EDG front end
  - Also includes other useful extensions (e.g., typeof)
- Standard Metacode Library
  - Fairly straightforward
- Metacode functions
  - IL interpreter: Cumbersome
  - Otherwise, much like inline functions

Implementation Notes (2)

- Metacode blocks
  - Not yet implemented
  - Like metacode functions, but larger impact on syntax
- Metacode injection
  - Not yet implemented
  - Expected to be relatively hard
  - Some similarities to template instantiation
Open Issues/To Do

- Nonlocal metacode variables?
- User-defined metacode types?
- Exported metacode functions?
- Design of Standard Metacode Library?
- Complete EDG-based implementation
- Alternative implementation
- Metacode debugging tools
- Users needed

Contact Info

metacode@vandevoorle.com

http://vandevoorle.com
http://www.edg.com

Daveed Vandevoorle
289 Kinnelon Road
Kinnelon, NJ 07405
U.S.A.